Open Street Map Case Study

by Marta Alonso

1. Area of study

City of Vigo (Pontevedra) - Spain -> https://www.openstreetmap.org/relation/341381 (https://www.openstreetmap.org/relation/341381)

This is my hometown, so, I want to explore what interesting information OSM has about my city.

2. Download and analyze the dataset

Data is downloaded with the script below using the Overpass API (http://overpass-api.de/query_form.html):

```
[out:xml];
(
  node(42.1125, -8.9597, 42.2847, -8.5855);
  <;
);
out meta;
```

The datafile size is 59.9MB, let´s have a look at how many elements, nodes and ways are contained in the file:

In [2]:
```
import xml.etree.cElementTree as ET
import pprint
from collections import defaultdict

#number of nodes and ways
def count_fixme_points(filename):
    result = {"node": 0,
              "way": 0,
              "total_elements" : 0
             }
    for event, elem in ET.iterparse(filename, events=("start",)):
        result["total_elements"] += 1
        if elem.tag == "node":
            result["node"] += 1
        elif elem.tag == "way":
            result["way"] += 1
    return result

result = count_fixme_points("vigo_box.osm")
pprint.pprint(result)
```

```
{'node': 269364, 'total_elements': 766760, 'way': 29258}
```

We have a total of 766760 elements with 269364 nodes and 29258 ways.

3. Auditing data

Is the data accurate?

We only want to take into account data that is reliable, hence we want to check all the fixme tags, and see what comments they include. (The code from this chapter is included in the **audit_fixme.py** script):

In [3]:
```
# Code to create a dictionary with the different "fixme"
# comments and the number of appearances differente comments have.

import xml.etree.cElementTree as ET
import pprint
from collections import defaultdict

def fixme_points(filename):
    dict = defaultdict(int)
    for event, elem in ET.iterparse(filename, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                if (tag.attrib["k"] == "fixme"):
                    dict[tag.attrib["v"].lower().strip()] += 1
    return dict

cities = fixme_points('vigo_box.osm')
pprint.pprint(cities)
```

```
defaultdict(<class 'int'>,
            {'*': 7,
             'averiguar nome': 1,
             'codigo=o99, otras actividades. afinar office=x si es posi
ble.': 7,
             'comprobar que es correcto': 131,
             'comprobar que es correcto el etiquetado': 1,
             'comprobar si es frutas beni o frutería beni': 1,
             'comprobar si es sea parking público o al aire libre. en c
aso de serlo debería ser amenity= parking.': 1,
             'comprobar sobre terreno': 4,
             'la mitad hacia el oeste puede ser el 44 de avenida de gal
icia': 2,
             'maxheight': 3,
             'name': 1,
             'name,operator': 1,
             'possible wrong housenumber': 1,
             "what's that?": 1})
```

Most of these "fixme´d" data points say "check for correctness" or other comments meaning that the data could be inaccurate. Therefore, we will drop these data points, we need a function that solves whether data points need to be fixed/double checked or not, the function would be the one below:

In [4]:
```
def is_reliable(elem):
    for tag in elem.iter("tag"):
        if (tag.attrib["k"] != "fixme"):
            return True
```

How many of the points are reliable and how many are not?

In [5]:
```
def count_fixme_points(filename):
    result = {"fixme": 0,
              "reliable": 0
             }
    for event, elem in ET.iterparse(filename, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            found = False
            for tag in elem.iter("tag"):
                if (tag.attrib["k"] == "fixme"):
                    found = True
            if found:
                result["fixme"] += 1
            else:
                result["reliable"] += 1
    return result

result = count_fixme_points("vigo_box.osm")
pprint.pprint(result)
```

```
{'fixme': 162, 'reliable': 298460}
```

Well, we have a total of 298460 reliable points and 162 points that need to be double checked. We will drop those 162.

Where is the data from?

We want to use only data from the city of Vigo, so we need to audit the city names of the data points. We will use the function below (the code for this chapter is included in the **audit_cities.py** script):

In [6]:
```
# Code to determine which city the data points belong to

import xml.etree.cElementTree as ET
import pprint
from collections import defaultdict

def city_names(filename):
    dict = defaultdict(int)
    for event, elem in ET.iterparse(filename, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            if is_reliable(elem):
                for tag in elem.iter("tag"):
                    if (tag.attrib["k"] == "addr:city"):
                        dict[tag.attrib["v"]] += 1
    return dict

cities = city_names('vigo_box.osm')
pprint.pprint(cities)
```

```
defaultdict(<class 'int'>,
            {'BAIONA': 5,
             'Baiona': 17,
             'Cangas': 85,
             'Cans': 1,
             'Gondomar': 6,
             'Moaña': 5,
             'Mos': 15,
             'Nigran': 6,
             'Nigrán': 5,
             'Nigrán Priegue, Pontevedra': 1,
             'O Cruceiro - Cedeira - Redondela': 1,
             'O Porriño': 132,
             'Porriño (O)': 1,
             'Redondela': 16,
             'Sanguiñeda': 1,
             'Sanguiñeda - Mos': 1,
             'Valladares Vigo': 1,
             'Vigo': 964,
             'cangas': 2,
             'vigo': 9})
```

We have discovered two problems here, since we are using a box to download Vigo´s data, we are getting also many nodes from other cities, but not only that, we have upper and lower caps for Vigo in several nodes, we need to take care of this.

We create a function to check if the node belongs to Vigo and to convert the city names to lowercase to homogenize those values. Apart from the "addr_city" attribute, we include in the search the "is_in:city" and "is_in:municipality" since we have realized any of them could specify which city the element belongs to:

In [7]:
```
# Function to determine if a xml element
# belongs to a specified city or not

def is_my_city(elem, city):
    for tag in elem.iter("tag"):
        if ((tag.attrib["k"] == "addr:city") or
            (tag.attrib["k"] == "is_in:city") or
            (tag.attrib["k"] == "is_in:municipality")):
            if (city == tag.attrib["v"].lower()):
                return True
    return False
```

Are street names homogeneous?

We want to audit the accuracy and uniformity of the street names. All of them should include any of the expected street types in Galician language (rúa, avenida, camiño, etc). We use the following code to check unexpected street types in our dataset. The code for this chapter is included in the script **audit_streets.py**.

In [8]:
```
# Code to determine which street types are included in the dataset
# others than the expected ones in the "expected" array.

import xml.etree.cElementTree as ET
from collections import defaultdict
import re
import pprint

OSMFILE = "vigo_box.osm"
street_type_re = re.compile(r'^\S+\.?\b', re.IGNORECASE)


expected = ["Rúa", "Avenida", "Camiño", "Praza", "Lugar", "Estrada"]

def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)


def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            if (is_reliable(elem) and is_my_city(elem, "vigo")):
                for tag in elem.iter("tag"):
                    if is_street_name(tag):
                        audit_street_type(street_types, tag.attrib['v'])
    osm_file.close()
    return street_types

st_types = audit(OSMFILE)
pprint.pprint(dict(st_types))
```

```
{'1ª': {'1ª Travesía do Couto de San Honorato'},
 'AVDA': {'AVDA DE MADRID',
          'AVDA. MADRID',
          'AVDA. RICARDO MELLA, S/N (PO-325 KM 5.8)'},
 'AVENIDA': {'AVENIDA BEIRAMAR',
             'AVENIDA GALICIA-TEIS',
             'AVENIDA MADRID, S/N',
             'AVENIDA RICARDO MELLA, S/N'},
 'Alcalde': {'Alcalde Gregorio Espino'},
 'Aragon': {'Aragon'},
 'Aragón': {'Aragón'},
 'As': {'As Lagoas, Marcosende'},
 'Atlantida': {'Atlantida'},
 'Baixada': {'Baixada a Rúa Conde de Torrecedeira'},
 'Balaidos': {'Balaidos'},
 'Bao': {'Bao'},
 'Bembrive': {'Bembrive'},
 'Buenos': {'Buenos Aires'},
 'C/Alcalde': {'C/Alcalde de Lavadores'},
 'CALLE': {'CALLE TOMAS PAREDES'},
 'CARRETERA': {'CARRETERA MADRID - VIGO KM 663.5',
               'CARRETERA VIGO- MADRID KM. 663'},
 'CL': {'CL AVENIDA DE MADRID',
        'CL REPUBLICA ARGENTINA 8',
        'CL TRAVESIA DE VIGO'},
 'Calle': {'Calle C'},
 'Cambados': {'Cambados'},
 'Camelias': {'Camelias'},
 'Camino': {'Camino da Seara', 'Camino da Galindra', 'Camino Camilo Tuc
he'},
 'Camposancos': {'Camposancos'},
 'Campus': {'Campus de Vigo, Lagoas-Marcosende'},
 'Canovas': {'Canovas Del Castillo'},
 'Castelao': {'Castelao'},
 'Castrelos': {'Castrelos'},
 'Cesareo': {'Cesareo Vázquez'},
 'Costa': {'Costa'},
 'Couto': {'Couto Piñeiro'},
 'Cronista': {'Cronista Rodríguez Elias'},
 'E': {'E. Mtnez Garrido-Alcalde'},
 'Ecuador': {'Ecuador'},
 'Enrique': {'Enrique Lorenzo'},
 'Estorniño': {'Estorniño'},
 'Eugenio': {'Eugenio Fadrique'},
 'Faisan': {'Faisan'},
 'Fonte': {'Fonte das Abelleiras, Campus Universitario de Vigo'},
 'Galicia': {'Galicia'},
 'García': {'García Barbon'},
 'Girona': {'Girona'},
 'Gonzalez': {'Gonzalez Sierra'},
 'Gran': {'Gran Via'},
 'Grove': {'Grove (O)'},
 'Hispanidade': {'Hispanidade'},
 'Illa': {'Illa de Toralla,  Coruxo'},
 'Illas': {'Illas Baleares', 'Illas Canarias'},
 'Jaime': {'Jaime Balmes'},
 'Jenaro': {'Jenaro De La Fuente'},
```

```
'Jose': {'Jose Gómez Posada-Curros'},
'Lagoas': {'Lagoas Marcosende (Campus Universitario)'},
'Laxe': {'Laxe'},
'Leonardo': {'Leonardo Alonso'},
'Macal': {'Macal'},
'Manuel': {'Manuel De Castro'},
'Marcosende': {'Marcosende'},
'Meixoeiro': {'Meixoeiro'},
'Miradoiro': {'Miradoiro (Do)'},
'Paseo': {'Paseo marítimo da ETEA'},
'Pescadores': {'Pescadores'},
'Pizarro': {'Pizarro'},
'Playa': {'Playa de Canido'},
'Plaza': {'Plaza de la Estrella '},
'Praza/patio': {'Praza/patio interior'},
'Principe': {'Principe'},
'Progreso': {'Progreso'},
'Provincial': {'Provincial'},
'Ramon': {'Ramon Nieto'},
'Ramón': {'Ramón del Valle-Inclán'},
'Republica': {'Republica Argentina'},
'Ricardo': {'Ricardo Mella'},
'Ronda': {'Ronda Don Bosco'},
'Rosalía': {'Rosalía de castro', 'Rosalía de Castro, 28'},
'Rotonda': {'Rotonda Zona Franca de Bouzas'},
'Rua': {'Rua Abeleira Menendez',
        'Rua Leonardo Da Vinci',
        'Rua Ramon Nieto',
        'Rua Salceda de Caselas',
        'Rua Salgado',
        'Rua Teofilo Llorente',
        "Rua de Alfonso X 'O Sabio'"},
'Salgueira': {'Salgueira'},
'Salvaterra': {'Salvaterra'},
'San': {'San Cristovo', 'San Roque'},
'Sanjurjo': {'Sanjurjo Badía', 'Sanjurjo Badia'},
'Santa': {'Santa Marina'},
'Senda': {'Senda do río Eifonso'},
'Senra': {'Senra De Abaixo'},
'Subida': {'Subida Relfas'},
'TRAVESIA': {'TRAVESIA DE VIGO'},
'Teixugueiras': {'Teixugueiras (As)'},
'Toledo': {'Toledo'},
'Tomás': {'Tomás A Alonso', 'Tomás Paredes'},
'Travesia': {'Travesia De Vigo', 'Travesia de Jacinto Benavente'},
'Travesía': {'Travesía de Vigo', 'Travesía do Príncipe'},
'Urzaiz': {'Urzaiz'},
'Valadares': {'Valadares'},
'Venezuela': {'Venezuela'},
'Via': {'Via Norte'},
'Zamora': {'Zamora'}}
```

As we can see the street types and names are far from homogenous, there are several problems with them:

- written inconsistencies: some street names are written both with accent marks and without them,
- language inconsistencies: there are street types like "street" or "square" written both in Spanish and in Galician (regional language) so the same street could appear twice written differently (calle/rua, plaza/praza)
- case inconsistencies: the same street is written some times in upper and sometimes in lower cases
- names like street(calle/rua) are sometimes abbreviated (calle, cl, c/)

To solve these issues we need the following functions that are included in the script **audit_streets.py**. As shown below the list of expected street types was extended and homogenized in lower caps and without accent marks:

In [9]:
```
# Functions to clean and homogenize the street names and types

expected = ["rua", "avenida", "camiño", "praza", "lugar", "estrada", "ca
leixon", "poligono", "area", "paseo"]

mapping = { "avda": "avenida",
            "calle": "rua",
            "cl" : "rua",
            "carretera" : "estrada",
            "ctra" : "estrada",
            "calleja/callejón" : "caleixon",
            "plaza" : "praza",
            "praza/patio" : "praza/patio",
            "C/Alcalde" : "rua alcalde"
          }

def delete_accents(word):
    return word.replace(u"á", u"a").replace(u"é", u"e").replace(u"í", u
"i").replace(u"ó", u"o").replace(u"ú", u"u")
    #return word

def get_street(elem):
    city = None
    for tag in elem.iter("tag"):
        if (tag.attrib["k"] == "addr:street"):
            street = delete_accents(tag.attrib["v"].lower().strip())
            m = street_type_re.search(street)
            if m:
                street_type = m.group()
            if street_type in expected:
                return street
            elif street_type in mapping.keys():
                street = street.replace(street_type, mapping[street_type
])
                return street
            else:
                return "rua "+street
    return city
```

4. Cleaning data

We are going to read the data from the osm file, clean it with the different functions we have created in the previous chapters, and transform it to a json file to be imported into MongoDB. The format of the JSON is going to be similar to the one used in the course case of study. Following the same rules:

- only 2 types of top level tags will be processed: "node" and "way"
- all attributes of "node" and "way" will be turned into regular key/value pairs, except:
  - attributes in the CREATED array that will be added under a key "created"
  - attributes for latitude and longitude that will be added to a "pos" array.
- if the second level tag "k" value contains problematic characters, it will be ignored
- if the second level tag "k" value starts with "addr:", it will be added to a dictionary "address"
- if the second level tag "k" value does not start with "addr:", but contains ":", the colon will be converted to a valid character like "_".
- if there is a second ":" that separates the type/direction of a street, the tag will be ignored
- for "way" specifically:

  ```
  <nd ref="305896090"/>
  <nd ref="1719825889"/>
  ```

  will be turned into

  ```
  "node_refs": ["305896090", "1719825889"]
  ```

  An example of the new JSON shape is shown below:

```
{
"id": "2406124091",
"type": "node",
"visible":"true",
"created": {
          "version":"2",
          "changeset":"17206049",
          "timestamp":"2013-08-03T16:43:42Z",
          "user":"linuxUser16",
          "uid":"1219059"
        },
"pos": [41.9757030, -87.6921867],
"address": {
          "housenumber": "5157",
          "postcode": "60625",
          "street": "North Lincoln Ave"
        },
"amenity": "restaurant",
"cuisine": "mexican",
"name": "La Cabana De Don Luis",
"phone": "1 (773)-271-5176"
}
```


The code shown below can be found at **data_process.py** script.

In [10]:
```
# Code to process the xml file, cleanint it a transforming into JSON

#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.cElementTree as ET
import pprint
import re
import codecs
import json
#from audit_fixme import is_reliable
#from audit_cities import is_my_city
#from audit_streets import get_street

lower = re.compile(r'^([a-z]|_)*$')
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')


def shape_element(element):
    node = {}
    if element.tag == "node" or element.tag == "way" :
        if is_reliable(element) and is_my_city(element, "vigo"):
            node['id'] = element.attrib.get('id')
            node['type'] = element.tag
            node['visible'] = element.attrib.get('visible')
            node['created'] = {}
            node['created']['version'] = element.attrib.get('version')
            node['created']['changeset'] = element.attrib.get('changese
t')
            node['created']['timestamp'] = element.attrib.get('timestam
p')
            node['created']['user'] = element.attrib.get('user')
            node['created']['uid'] = element.attrib.get('uid')
            if element.tag == "node" :
                node['pos'] = [float(element.attrib.get('lat')), float(e
lement.attrib.get('lon'))]
            if element.tag == "way" :
                node['node_refs'] = []
                for nd in element.iter('nd'):
                    node['node_refs'].append(nd.attrib['ref'])
            for tag in element.iter('tag'):
                if problemchars.search(tag.attrib['k']):
                    pass
                elif lower.search(tag.attrib['k']):
                    node[tag.attrib['k']] = tag.attrib['v']
                else:
                    m = lower_colon.search(tag.attrib['k'])
                    if m:
                        if len(m.group().split(":")) != 2 :
                            pass
                        elif m.group().split(":")[0] == "addr":
                            if "address" not in node.keys():
                                node['address'] = {}
                            if m.group().split(":")[1] == "street":
                                node['address']['street'] = get_street(e
lement)
```

```
                            else:
                                node['address'][m.group().split(":")[1]]
 = tag.attrib['v']
                        else:
                            new = tag.attrib['k'].replace(":", "_")
                            node[new] = tag.attrib['v']

        return node
    else:
        return None


def process_map(file_in, pretty = False):
    # You do not need to change this file
    file_out = "{0}.json".format(file_in)
    data = []
    with codecs.open(file_out, "w") as fo:
        for _, element in ET.iterparse(file_in):
            el = shape_element(element)
            if el:
                data.append(el)
                if pretty:
                    fo.write(json.dumps(el, indent=2)+"\n")
                else:
                    fo.write(json.dumps(el) + "\n")
    return data

data = process_map('vigo_box.osm', False)
```

5. Working with MongoDB

Importing into DB

The json file we have just created is importend into MongoDB with the following command:

```
mongoimport -d osm_db -c vigo_col --file vigo_box.osm.json
```

Using console to query the DB

We connect to the MongoDB console and launch several queries

```
$Project> mongo
MongoDB shell version: 3.0.15
connecting to: test
> use osm_db
switched to db osm_db
```

Let´s count the number of nodes and ways in the DB

```
> db.vigo_col.find({"type":"node"}).count()
580
> db.vigo_col.find({"type":"way"}).count()
439
```

We have 580 nodes and 439 ways in our DB.

```
> db.vigo_col.distinct("created.uid").length
147
```

There are 147 unique users who have contributed to that data from Vigo city.

Let´s do some aggregation queries:

```
> db.vigo_col.aggregate([{"$group" : {"_id" : "$amenity", "count": {"$sum" :
 1}}},{"$sort" : {"count" : -1}}, {"$limit" : 10}])
{ "_id" : null, "count" : 678 }
{ "_id" : "pharmacy", "count" : 112 }
{ "_id" : "cafe", "count" : 36 }
{ "_id" : "restaurant", "count" : 33 }
{ "_id" : "bank", "count" : 30 }
{ "_id" : "bar", "count" : 23 }
{ "_id" : "school", "count" : 22 }
{ "_id" : "fuel", "count" : 16 }
{ "_id" : "fast_food", "count" : 8 }
{ "_id" : "place_of_worship", "count" : 6 }
```

We have a list of the 10 most common amenities, been the first one in the list the number of elements without an amenity tag. We can see that pharmacies are the most common one and cafes, restaurants and banks are the second most common amenities in the dataset.

```
> db.vigo_col.aggregate([{"$match" : {"amenity" : "pharmacy"}},{"$group" :
 {"_id" : "$address.street", "count": {"$sum" : 1}}},{"$sort" : {"count" : -
1}}, {"$limit" : 5}])
{ "_id" : "rua ramon nieto", "count" : 6 }
{ "_id" : "rua travesia de vigo", "count" : 6 }
{ "_id" : "rua urzaiz", "count" : 6 }
{ "_id" : "rua sanjurjo badia", "count" : 5 }
{ "_id" : "rua teixugueiras (as)", "count" : 3 }

> db.vigo_col.aggregate([{"$match" : {"amenity" : "cafe"}},{"$group" : {"_i
d" : "$address.street", "count": {"$sum" : 1}}},{"$sort" : {"count" : -1}},
 {"$limit" : 5}])
{ "_id" : "avenida da florida", "count" : 6 }
{ "_id" : "rua travesia de vigo", "count" : 4 }
{ "_id" : "rua de venezuela", "count" : 3 }
{ "_id" : "rua puerto rico", "count" : 2 }
{ "_id" : "avenida da gran via", "count" : 2 }
```

With the last two queries we wanted to find the top 5 streets with the biggest numbers of pharmacies and cafes. In the case of pharmacies that results make a lot of sense since those streets are some of the longest streets in Vigo. For the cafe´s query, the streets in the results are long streets situated in the city centre which also make a lot of sense. Except for the case of the Street Puerto Rico which is in the city center but it is much smaller than the other streets in the top 5.

6. Future improvements

This was just an example of what can be done cleaning and auditing a dataset, but we could go further, auditing and cleaning other fields in the dataset. For example we could audit a cross field constraint like cities and postal codes. Let´s make some exploring queries about postal codes:

```
> db.vigo_col.find({"address.postcode":{"$exists":1}}).count()
897
> db.vigo_col.find({"address.postcode":{"$exists":0}}).count()
122
```

We can see that from the 1019 elements in our filtered dataset, 897 have the tag "address.postcode" and 122 don´t. Let´s try to match now the postal codes of Vigo with the ones in the dataset:

```
> db.vigo_col.find({"address.postcode":{"$in":["36201", "36202", "36203", "3
6204", "36205", "36206", "36207", "36208", "36209", "36210", "36211", "3621
2", "36213", "36214", "36215", "36216", "36312", "36317", "36331"]}}).count
()
871
```

From those 897 with postal code, 871 are postalcodes that indeed match the existing postal codes in Vigo. What happen with the remaining 26?

```
> db.vigo_col2.find({"address.postcode":{"$nin":["36201", "36202", "36203",
 "36204", "36205", "36206", "36207", "36208", "36209", "36210", "36211", "36
212", "36213", "36214", "36215", "36216", "36312", "36317", "36331"],"$exist
s":1}},{"_id":0,"address.postcode":1})
{ "address" : { "postcode" : "36314" } }
{ "address" : { "postcode" : "36314" } }
{ "address" : { "postcode" : "36315" } }
{ "address" : { "postcode" : "36025" } }
{ "address" : { "postcode" : "36200" } }
{ "address" : { "postcode" : "36390" } }
{ "address" : { "postcode" : "36314" } }
{ "address" : { "postcode" : "36330" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36314" } }
{ "address" : { "postcode" : "36390" } }
{ "address" : { "postcode" : "36318" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36315" } }
{ "address" : { "postcode" : "36315" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36318" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36200" } }
{ "address" : { "postcode" : "36314" } }
{ "address" : { "postcode" : "36330" } }
{ "address" : { "postcode" : "36330" } }
{ "address" : { "postcode" : "36392" } }
{ "address" : { "postcode" : "36310" } }
{ "address" : { "postcode" : "36350" } }
```

These are the 26 postal codes that do not belong to Vigo, some of them, like the 36200 or 36025, are probably manual mistakes (36025 is probably 36205). A further investigation could analyze which cities those postal codes belong, if they really exists in other cities, or whether they are close to Vigo or not. The mistake could be in the postal code itself of in the city that the creator of the element wrote. Analyzing the content of the elements, name of the place and other fields would help to determine which of the mistakes happened in each case.

Going a little bit further, the location coordinates could be used to determine where exactly the point is, and which city and postal code it belongs to, to determine where the mistake was made. The problem in this kind of investigation is to choose, having the location, the city and the postal code, the right and the wrong field, we need always more information about the element, like name or description. Also if two out of three fields match that would help us to choose sides here.

REFERENCES

- https://wiki.openstreetmap.org/wiki/OSM_XML (https://wiki.openstreetmap.org/wiki/OSM_XML)
- https://wiki.openstreetmap.org/wiki/Elements (https://wiki.openstreetmap.org/wiki/Elements)
  https://wiki.openstreetmap.org/wiki/Map_features (https://wiki.openstreetmap.org/wiki/Map_features)
  https://wiki.openstreetmap.org/wiki/Key:is_in (https://wiki.openstreetmap.org/wiki/Key:is_in)
  http://nbconvert.readthedocs.io/en/5.x/install.html (http://nbconvert.readthedocs.io/en/5.x/install.html)
- https://stackoverflow.com/questions/13208286/how-to-write-latex-in-ipython-notebook (https://stackoverflow.com/questions/13208286/how-to-write-latex-in-ipython-notebook)
- http://data-blog.udacity.com/posts/2016/10/latex-primer/ (http://data-blog.udacity.com/posts/2016/10/latex-primer/)
- https://github.com/jupyter/help/issues/163 (https://github.com/jupyter/help/issues/163)
- https://docs.mongodb.com/manual/reference/ (https://docs.mongodb.com/manual/reference/)